



PyCell Studio Tutorial

L-2016.06

Copyright and Proprietary Information Notice

© 2016 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <http://www.synopsys.com/Company/Pages/Trademarks.aspx>.

All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

CONTENTS

Introduction.....	5
Python API.....	6
Anatomy of a Python Parameterized Cell.....	7
PyCell Development Process.....	7
IDE	7

Introduction

This tutorial explains the basics of the PyCell Studio product and how it can be used to develop parameterized cell layout generators. The first section of this tutorial introduces the technology for building parameterized cell generators. The next section of this tutorial describes the Integrated Development Environment (IDE) for developing and debugging Python programs. This IDE has been developed work with Python-based parameterized cells.

Introduction

Python Application Programming Interface (API) is used to create Python Parameterized Cells (PyCells) at any level of complexity that operate in the Open Access environment.

The **PyCell Studio™** is a group of tools, built around the API that can be used to form a complete PyCell™ development environment.

The groups of tools contained in the PyCell Studio are:

pyros™ – An Open Access layout viewer that has many additional built-in capabilities, including interactive DRC.

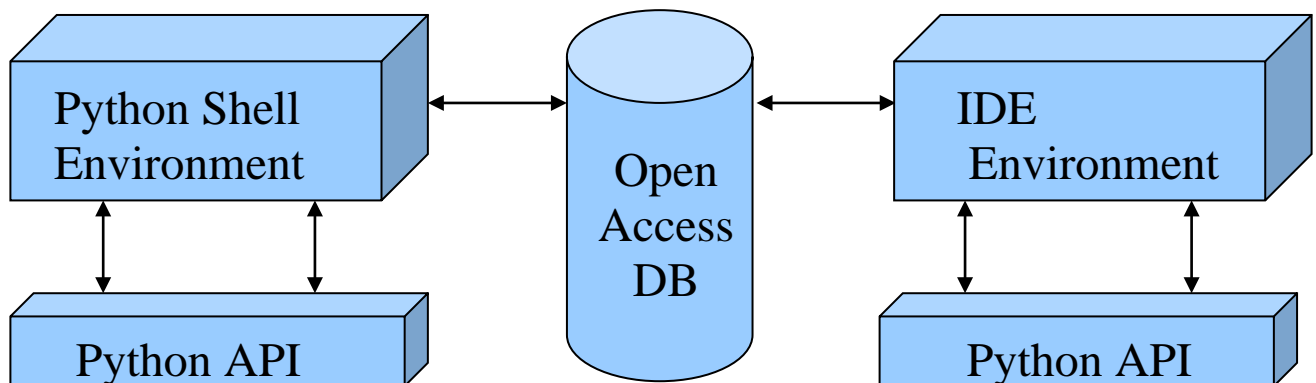
cnpy – A Python shell programming environment that can be used to interactively issue commands to the Python engine. **cnpy** can be used in conjunction with the **pyros** layout viewer to create a text based, development environment with graphic capability.

cnexp – A pre-configured version of the **cnpy** programming environment which has been tailored for learning the Python API.

cndbg – A fully functional Integrated Development Environment (IDE) with debugging capabilities. This IDE brings several benefits to the user both in the code development phase and in the debugging phase of a project and can also be used in conjunction with the **pyros** layout viewer to create a sophisticated PyCell development environment.

OA plug-in – The plug-in enables Open Access tools to execute the Python PyCell code and generate the geometries described in the code.

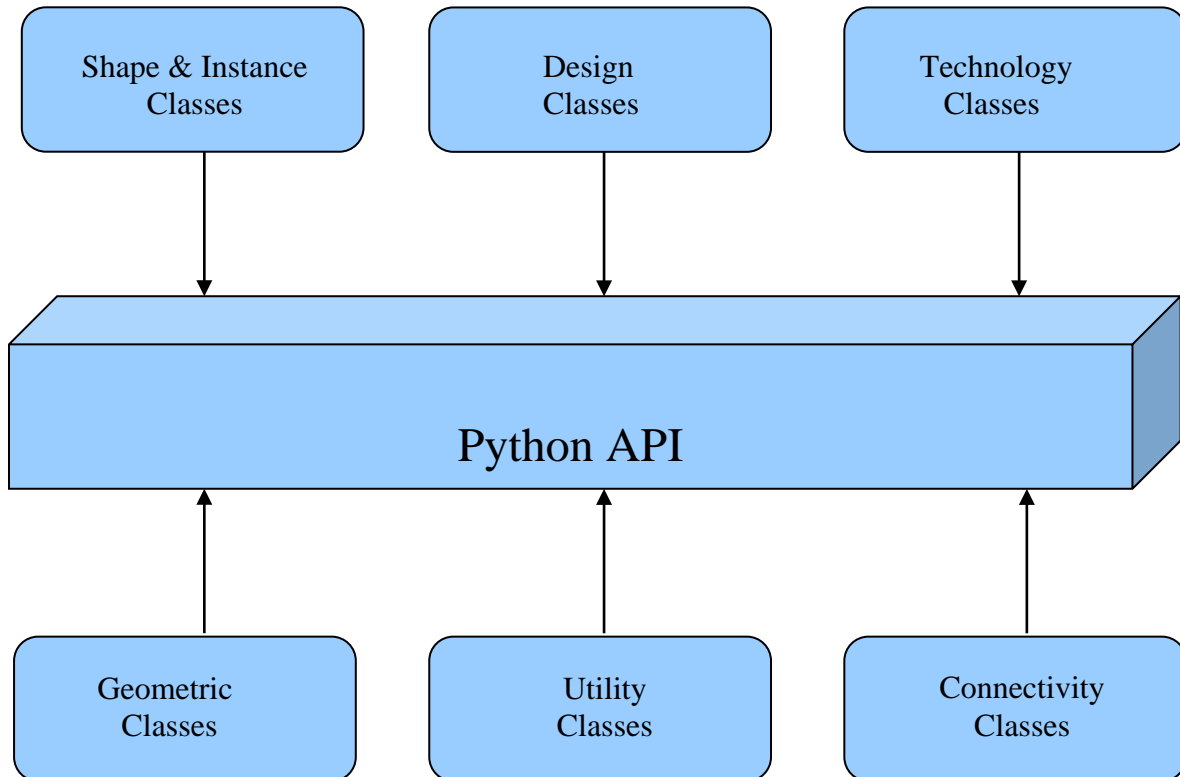
The complexity of the code being developed might dictate the environment used to create the code. For example, a novice user exploring the capabilities of the system or creating simple PyCells might use the shell environment, **cnpy** and the **pyros** layout viewer as a basic “high function, low frills” environment. A user that is creating a more complex PyCell might use the **cndbg Integrated Development Environment (IDE)** with the **pyros** layout viewer for its code management and graphic debugging capabilities.



Both environments offer complete access to the API. This tutorial will focus on the use of the IDE with the layout viewer for developing Python code. For more information on using the shell programming environment, please refer to the Python API Tutorial.

- ***Python API***

The Python API provides a large number of base and derived classes which are used to define the objects used to create Python Parameterized Cells (PyCells). This API is organized into functional groups as illustrated in the diagram below.



The Python programming language provides a highly productive object oriented programming environment which can be easily extended. The system extends these capabilities by providing class definitions for objects that closely correspond to the objects which are used in physical layout design. For example, layers, shapes on layers and other geometric objects are easily represented and acted on in this extended programming environment. Additionally, class definitions for higher level functions are provided that help the designer in more complex scenarios. These definitions include classes for contacts, connection bars, paths, routes and contact rings.

For a complete description of the Python API, please refer to the following documents available for download from the Synopsys website (<http://www.synopsys.com>)

Python API Tutorial
Python API Concise Reference Manual
Python API Reference Manual
Python API Summary

- ***Anatomy of a Python Parameterized Cell***

A PyCell, regardless of complexity, will have a minimum of three steps involved. These three steps are the following:

- 1) Defining parameters for the PyCell that can be used to customize each instance.
- 2) Reading and processing the parameters for the PyCell.
- 3) Generating the layout that is based on the defined and processed parameters.

Additional functions and steps that add different capabilities to the PyCell are normally included. Examples of these types of capabilities might include: functions to size the devices based on some user supplied algorithm, functions to create different device topologies based on input parameters and functions to place geometries based on complex, conditional design rules.

- ***PyCell Development Process***

There are four distinct steps that should be followed during the development of a PyCell:

- 1) Use the Python extensions to develop the source code for the PyCell. This code can be created using a text editor such as vi, or emacs, or the Integrated Development Environment. The code can either be created from scratch or by using code snippets from previously created PyCells or from the sample code provided.
- 2) Compile the Python code into an Open Access database. Synopsys supplies a utility, **cngenlib** which is used for this purpose.
- 3) View the generated layout. This can be done using the Open Access Viewer (**pyros**) or any OA compliant tool. The OA plug-in is required by the viewing tool.
- 4) Based on the results of viewing the generated layouts, make any necessary changes to the Python code, re-compile and view the resulting layout.

- ***IDE***

The Integrated Development Environment offers the user many capabilities which can be utilized during both the code development and the code debugging phases. The following are some of the debugging capabilities which are available in this built-in IDE debugging environment:

- Breakpoint definition -- Available from the Breakpoints menu, as well as by simply clicking on lines of source code in the code editor.

- Search -- Search and Replace facility is available from the Find menu, using dialogs to type in a search string, along with forward and backward searching. Can search and optionally replace code text strings as desired.
- Basic Debugging -- Set a breakpoint and start debugging. You can use the Step, Step Into and Step Out debugging commands to control the execution of source code through one or more pre-defined breakpoints. Separate debugger output window displays information about current stack state during process execution.
- Display of local and global variables -- During the debugging process, global and local variables are displayed, along with their dynamically updated values.
- Python Console -- This Python command prompt lets you try out code in a “sandbox” process which is kept isolated from the IDE and your debug process.

This tutorial will walk the user through the debugging process for a transistor PyCell using pre-existing blocks of code for specific functions. Synopsys supplies a library of elements which can be used as a starting point for code development. Once the code has been completed, then testing and debugging this PyCell code can be easily handled.

Please Note: Tutorial Setup Requirements

This tutorial assumes that you are running in the Linux environment and have already downloaded and installed the appropriate Linux PyCell Studio Kit, including the **pyros** layout viewing tool and the **cndbg** integrated IDE tool, and that the appropriate path information has been added to the user's environment.

This tutorial also assumes that you have copied the MyTutorialPyCells directory from the tutorial folder located in the installation hierarchy to your home directory. After copying this directory into your home directory, make sure that the files in this directory are writable; this is done with the Linux command “`chmod +w ~/MyTutorialPyCells/*`”. This tutorial will be run from your home directory for the Linux platform, while for the Windows platform, it will be run from the installation directory, so that this Python source code directory MyTutorialPyCells does not need to be copied.

In addition, if you do not have a “lib.defs” file in your home directory, then the “lib.defs” file located in ~/MyTutorialPyCells should be copied into the home directory for the Linux platform. If there already is a “lib.defs” file in the home directory, check that the following line is included in this “lib.defs” file:

```
DEFINE MyTutorialLib ~/MyTutorialLib
```

The PYTHONPATH environment variable contains the list of all Python modules which should be accessed by the Python interpreter when it is executing. For this tutorial, this PYTHONPATH environment variable should contain the directory path for the location of the Python source code directory containing the Python source code being debugged. This can be “.”, since this Python source code directory is located in the home directory.

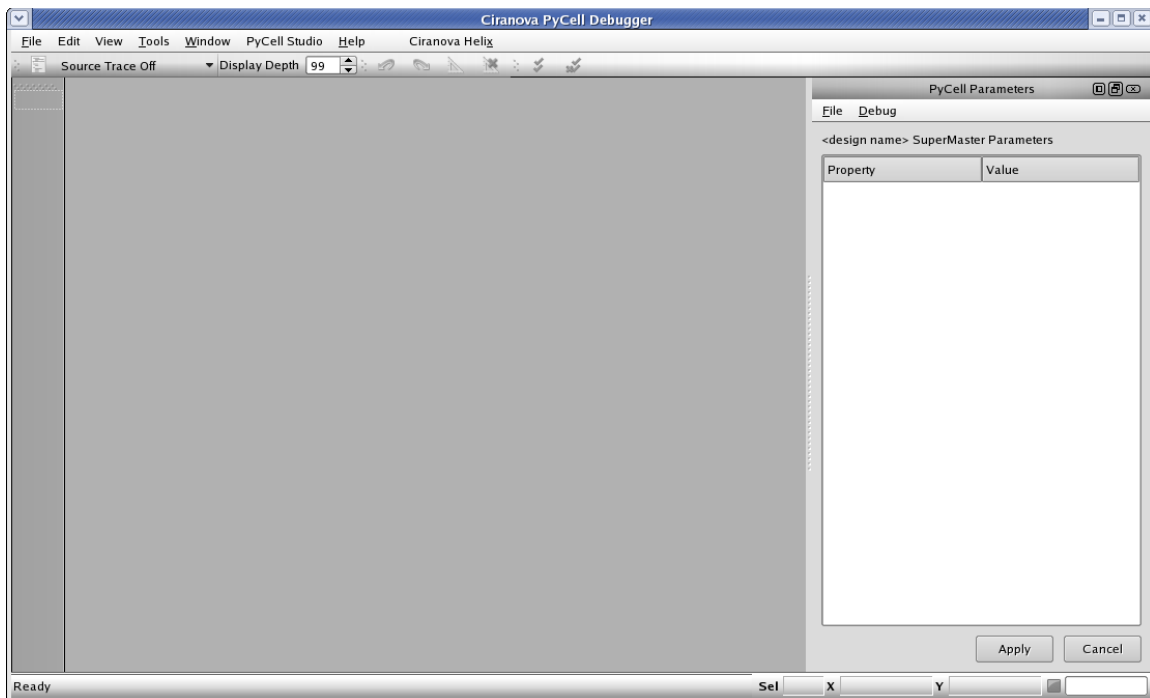
Commands that begin with “%” are executed from a shell.

Commands that begin with “(toolName)>>” are executed from a tool pull down menu.

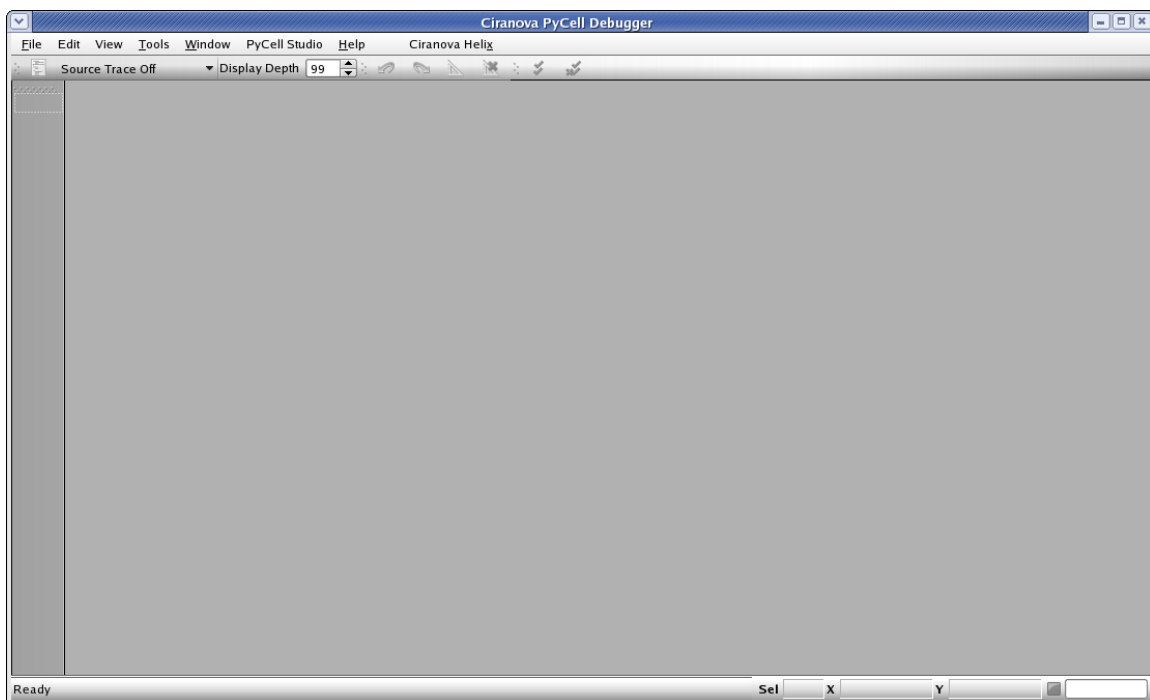
Step 1) Start the IDE

From the home directory, invoke the following command:

```
% cndbg &
```



This display should look like the above screen shot. Notice that the “PyCell Parameters” docking window is activated. The IDE is an environment which has several tools or docking windows which the user can choose to make available. To see the list of available docking windows, use the right mouse button in the toolbar/menu area to display the list of available panels (or windows); uncheck the “PyCell Parameters” checkbox, since we will not use this panel until later in this tutorial. Note that any of these docking can be resized as desired; this is done by “grabbing” the window boundary with the left mouse button and dragging it until the docking window is properly resized. At this point, the display should simply consist of the drawing canvas area, and look like the following screen shot:



Step 2) Setup Debugging for PyCell Project

A PyCell debugging project has the following properties:

OA Library Directory: Location of the Open Access Database that will be generated

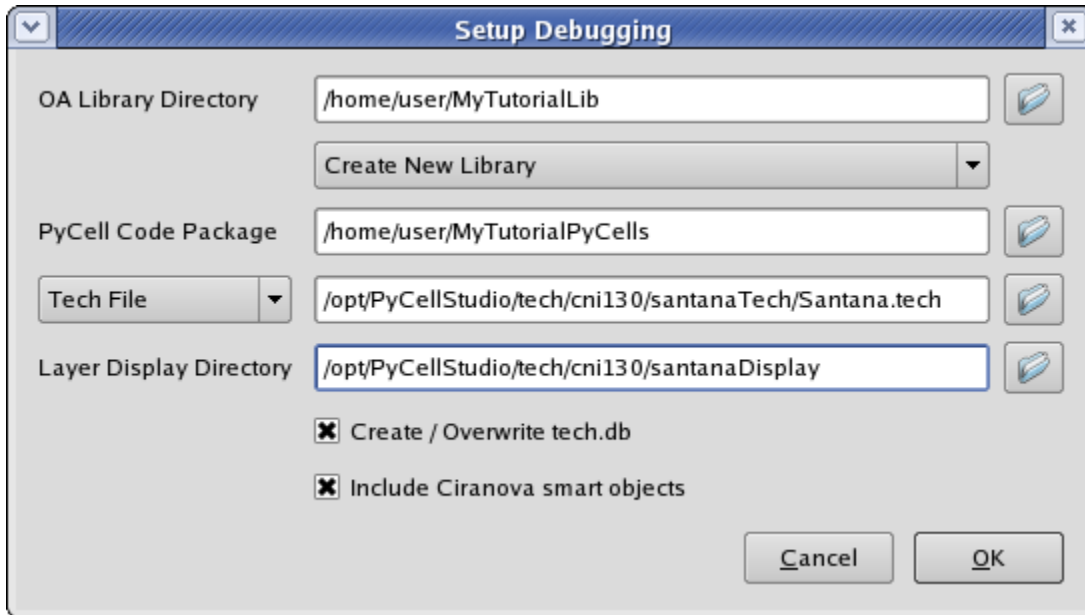
Python Source Directory: Location of Python source code which will be used to generate the Open Access Database. The directory path for the location of this Python source code directory should be contained in the PYTHONPATH environment variable. This can be “.”, since this Python source code directory is located in the home directory.

Tech File: Location and name of the technology file to be used for the project

Layer Display Directory: Location and name of the technology display directory which should be used for the project

PyCell Studio>>Debug Existing Code

From the main **cndbg** window, select the “Debug Existing Code” entry from the “PyCell Studio” menu entry, so that the “Setup Debugging” form is displayed:



Enter the following values for the fields in this dialog:

OA Library Directory = *~/MyTutorialLib*

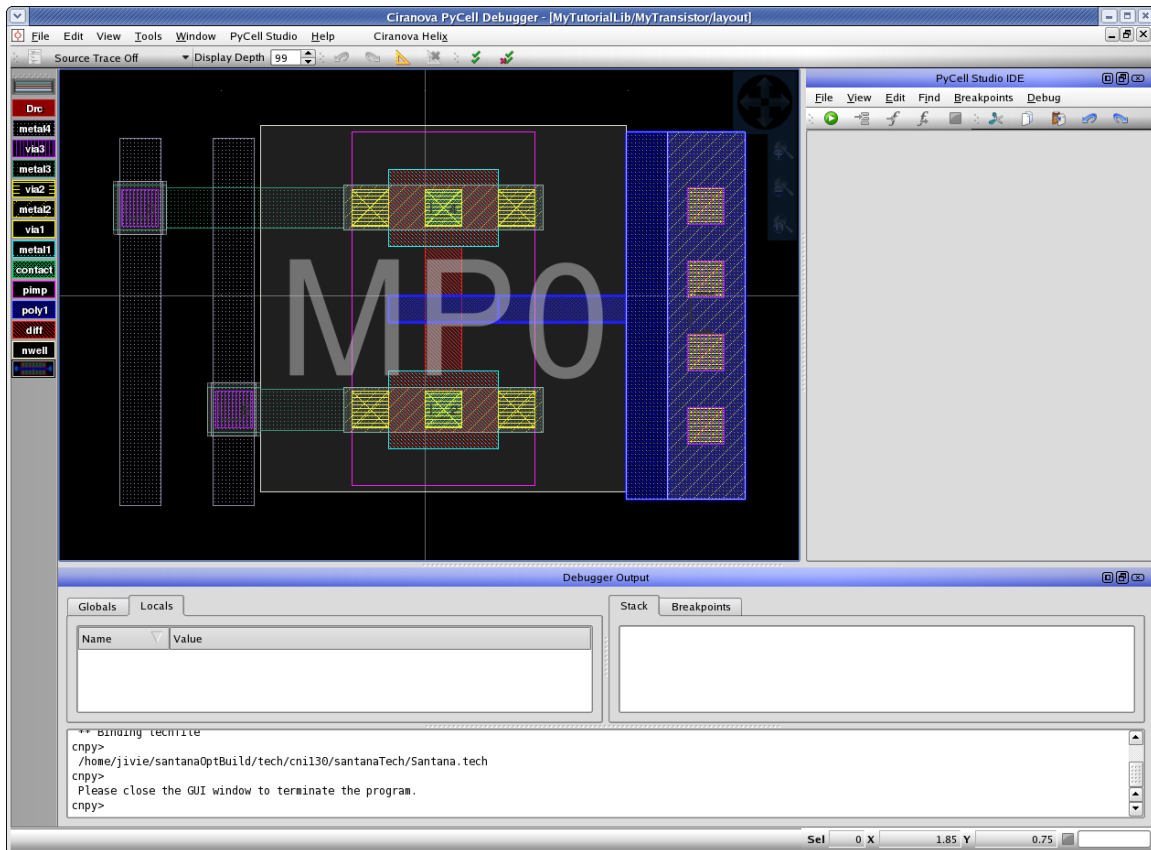
Python Source Directory = *~/MyTutorialPyCells*

Tech File = *\$CNI_ROOT/tech/cni130/santanaTech/Santana.tech*

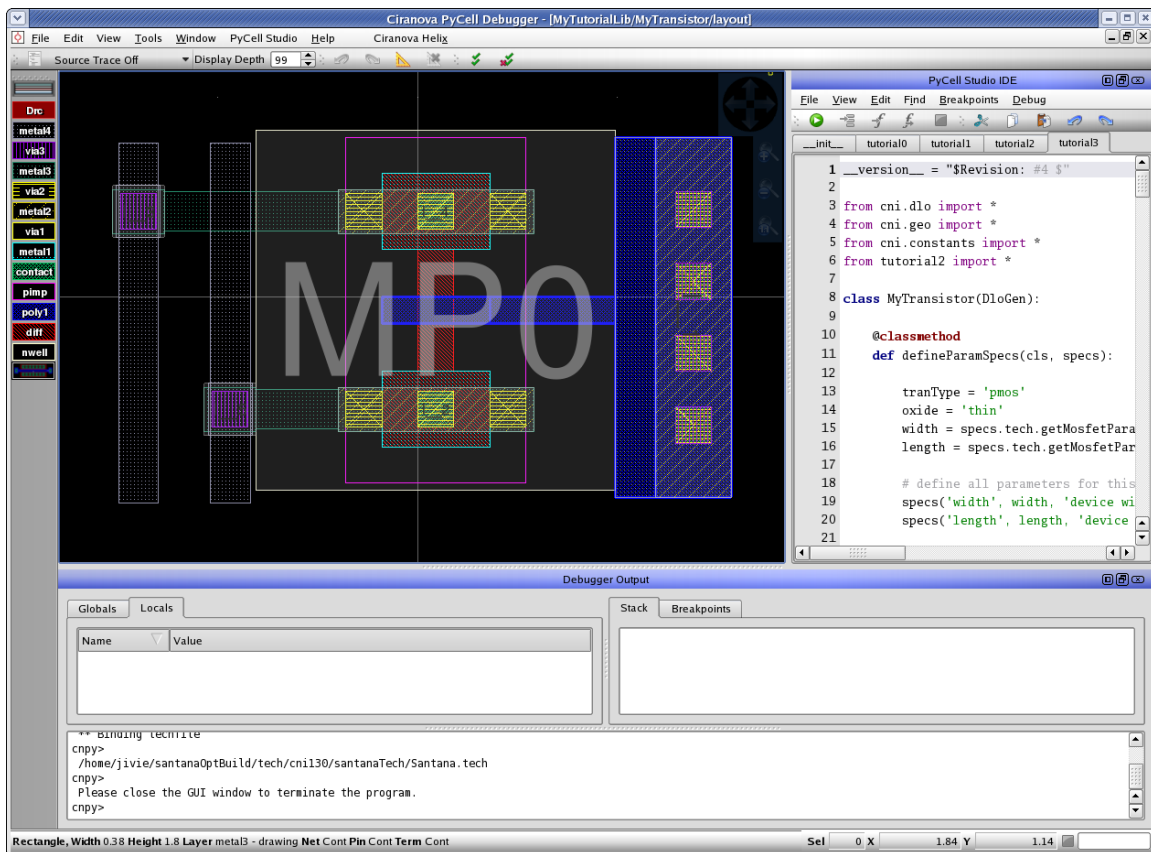
Layer Display Directory = *\$CNI_ROOT/tech/cni130/santanaDisplay*

Note that the CNI_ROOT environment variable points to the location of the installation; you should use the directory location which is specified by \$CNI_ROOT. Also, the “~” should be replaced by the actual location for the home directory.

Make sure that the “Create/Overwrite tech.db” and the “Include smart objects” checkboxes are both checked. Then click OK. After this form has been OK-ed, then the IDE debugger runs, and the OpenAccess PyCell library will be created, using the Python source code files specified for the “PyCell Code Package” field. In addition, the “Debugger Output” window will automatically be created and placed at the bottom of the display, and the “PyCell Studio IDE” window will be created and placed at the right side of the display. This display should then look like the following screen shot:

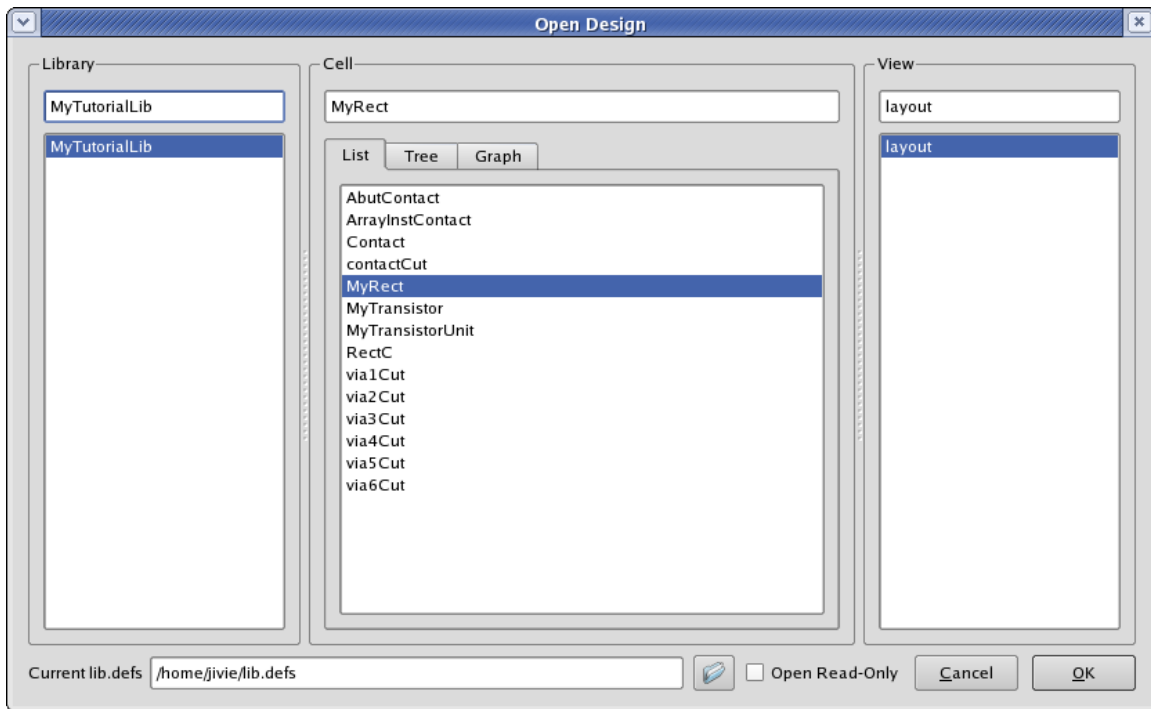


At this point, it is necessary to open the Python source code files for this OpenAccess PyCell library in the PyCell Studio IDE window. This is done by selecting the *File > Open File* menu entry in the “PyCell Studio IDE” docking window, for each of the Python files contained in the *MyTutorialPyCells* Python source code directory. Note that as each file is added to the “PyCell Studio IDE” docking window, a tab will be generated for that Python source code file. After all of these Python source code files have been added to the IDE window, then the display should look like the following screen shot:

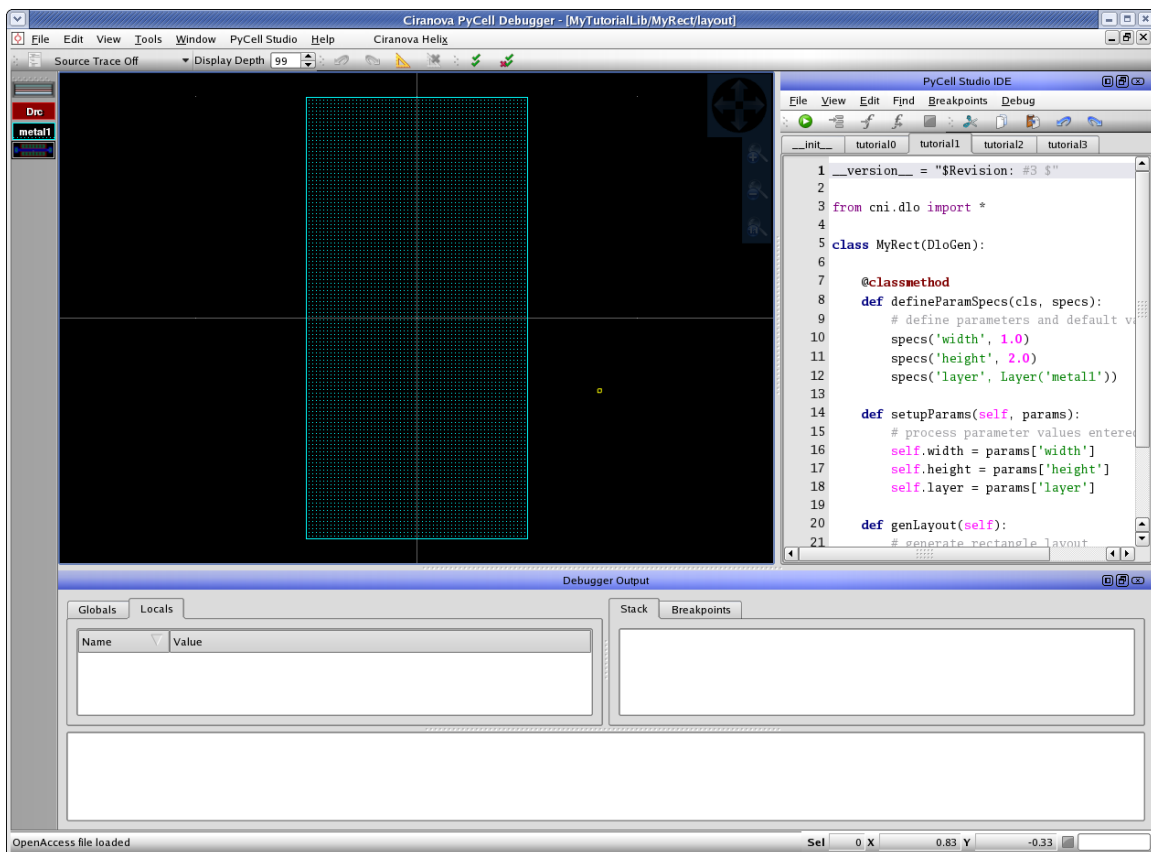


Step 3) Configure the windows

First select the tutorial1.py file in the “PyCell Studio IDE” docking window, by clicking on the filename tab “tutorial1” at the top of this window. Next, open the MyRect layout view, by using the *File>>Open* menu entry in the main **cndbg** window. Click to expand cell and view names, so that this dialog looks like the following:




The windows for the debugging session should then look like the following:



Step 4) Modify Python code and re-generate the library

To further illustrate the Python PyCell development process, we will change one of the default parameters and then rebuild the OpenAccess library.

In the “PyCell Studio IDE” docking window, change the default width value from 1.0 to 2.0; this is around line 10 in the file “tutorial1.py”, which contains the statement “specs('width', 1.0)”. Then use the *File>Save File* menu entry from the “PyCell Studio IDE” docking window to save this change to the Python source code file.

Now click on the *Debug* icon  on the tool bar in the “PyCell Studio IDE” docking window (or use the *Debug>Debug* menu entry), to execute **cngenlib** to generate the OpenAccess PyCell library. If it is not already open, then open the MyRect layout view using the *Window>Debugger Windows>1 MyTutorialLib/MyRect/layout* menu entry from the main **cndbg** window. Notice that the Parameterized Rectangle is now a different size, representing a square rectangle.

Step 5) Set breakpoints and step through Python code as it executes

In this step, we will set points (called breakpoints) in the Python source code at which execution of the code should pause. This important capability gives the PyCell code author access to the program as it is running where it is possible to both see the layout created up to that point in time and also full access to the program's variables. A breakpoint can be set by left clicking in the left margin of the source code text window.

We will now use the Transistor PyCell example, instead of the Rectangle PyCell example. Select the tutorial3.py Python file, which is the Python source code for this Transistor PyCell example, by clicking on the tab named “tutorial3” for this file.

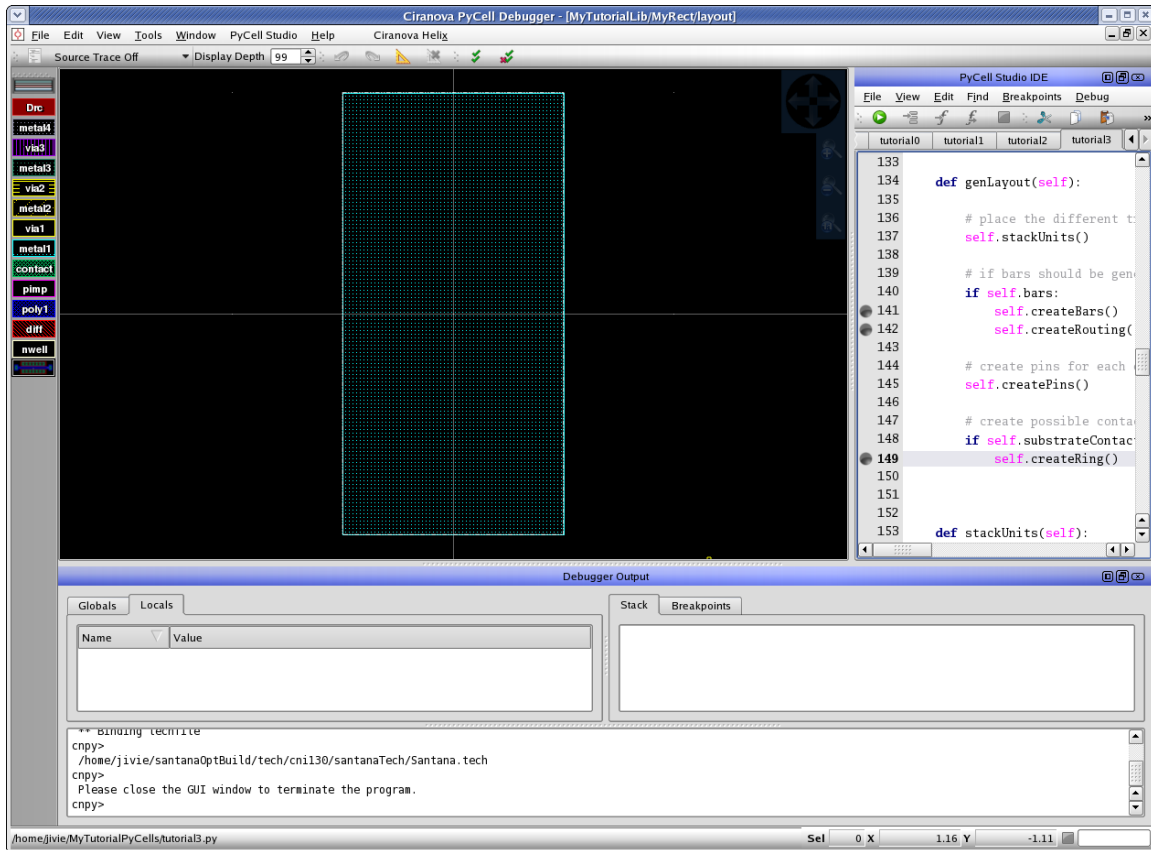
Locate the line which sets the default for substrate contact (around line 27) and modify this line of code, so that the default is set to True: specs('substrateContact',True).

Now locate the genLayout() function block of code. This function code is found around line 134 in the file tutorial3.py. This line of code can also be searched for by using the *Find>Find* menu entry from the “PyCell Studio IDE” docking window.

Within this genLayout() function, we want to set three breakpoints, at different points in the construction of the layout. Breakpoints should be added at the following functions:

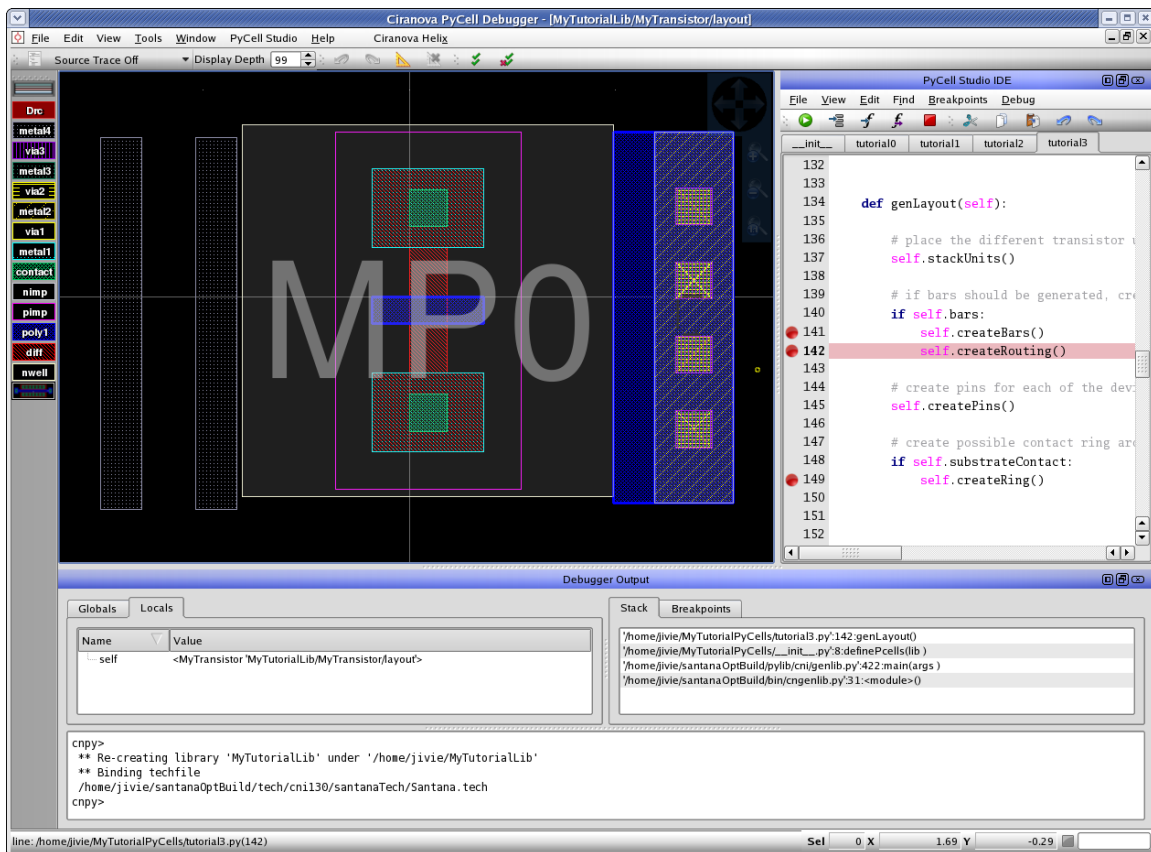
```
self.createBars()  
self.createRouting()  
self.createRing()
```

These breakpoints can be set by left clicking in the left margin of the source code text window on the three lines of code which call these three different functions. After this is done, then the screen should then appear as follows:



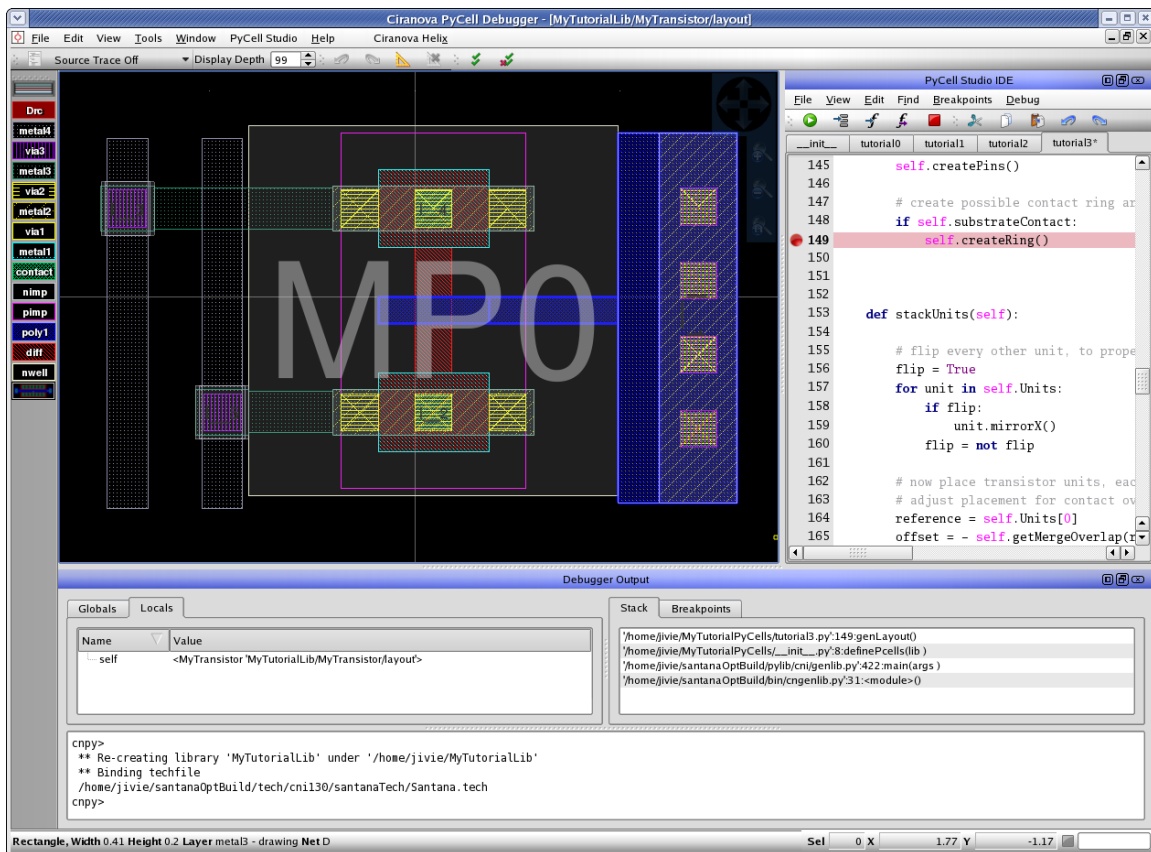
Use the *File>Save File* menu entry for the “PyCell Studio IDE” docking window to save the changes made to the Python source code file. Then click on the *Debug* toolbar icon in the “PyCell Studio IDE” docking window to start the debugging process. This time, **cndbg** will open the MyTransistor layout cell view. Use the mouse wheel (middle button) or the *View>ZoomHome* menu entry in the main **cndbg** window to set the viewing area. Notice that the program has just executed the `self.stackUnits()` function and has stopped at the first breakpoint.

Click on the *Debug* toolbar icon in the “PyCell Studio IDE” docking window to execute the code until the next breakpoint is reached. In this case, the `self.createBars()` function will have been executed. You should then see that three Bar objects have been created, as shown in the following screen shot:

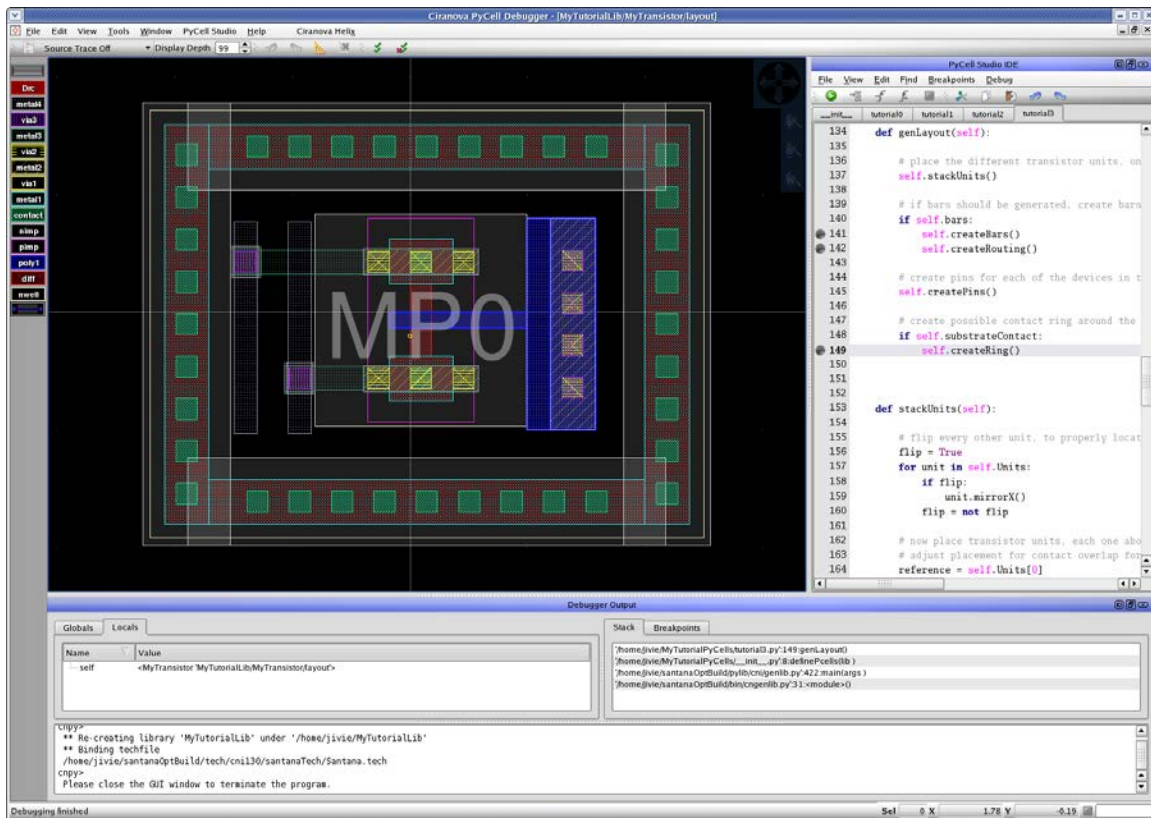


Use the (*PyCell Studio IDE*)>>*Step Into* icon to move into the `createRouting()` function in the code. After looking at the code for the routing functions, click on the second line in the “Stack” tab in the “Debugger Output” window to move back to the `self.createRouting()` breakpoint.

Click on the *Debug* toolbar icon in the “PyCell Studio IDE” docking window to continue execution to the next and final breakpoint. You should now see the transistor with the completed routing, as shown in the following screen shot:



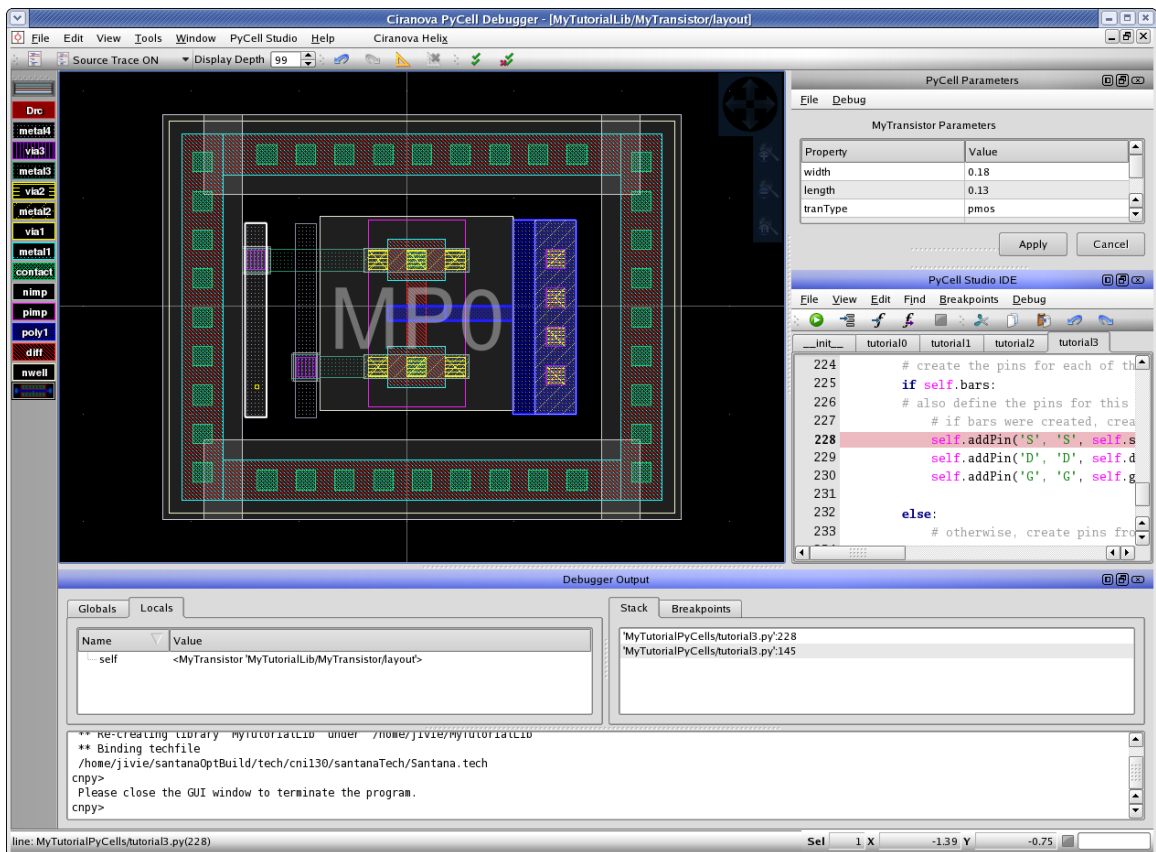
Now Click on the *Debug* toolbar icon in the “PyCell Studio IDE” docking window to continue execution past this final breakpoint. You should now see the completed transistor, with the generated contact ring, as shown in the following screen shot:



At this point, use the *Breakpoints>Clear All Breakpoints* menu entry from the “PyCell Studio IDE” docking window to remove the breakpoints which were set earlier. This is done to simplify the rest of this Tutorial.

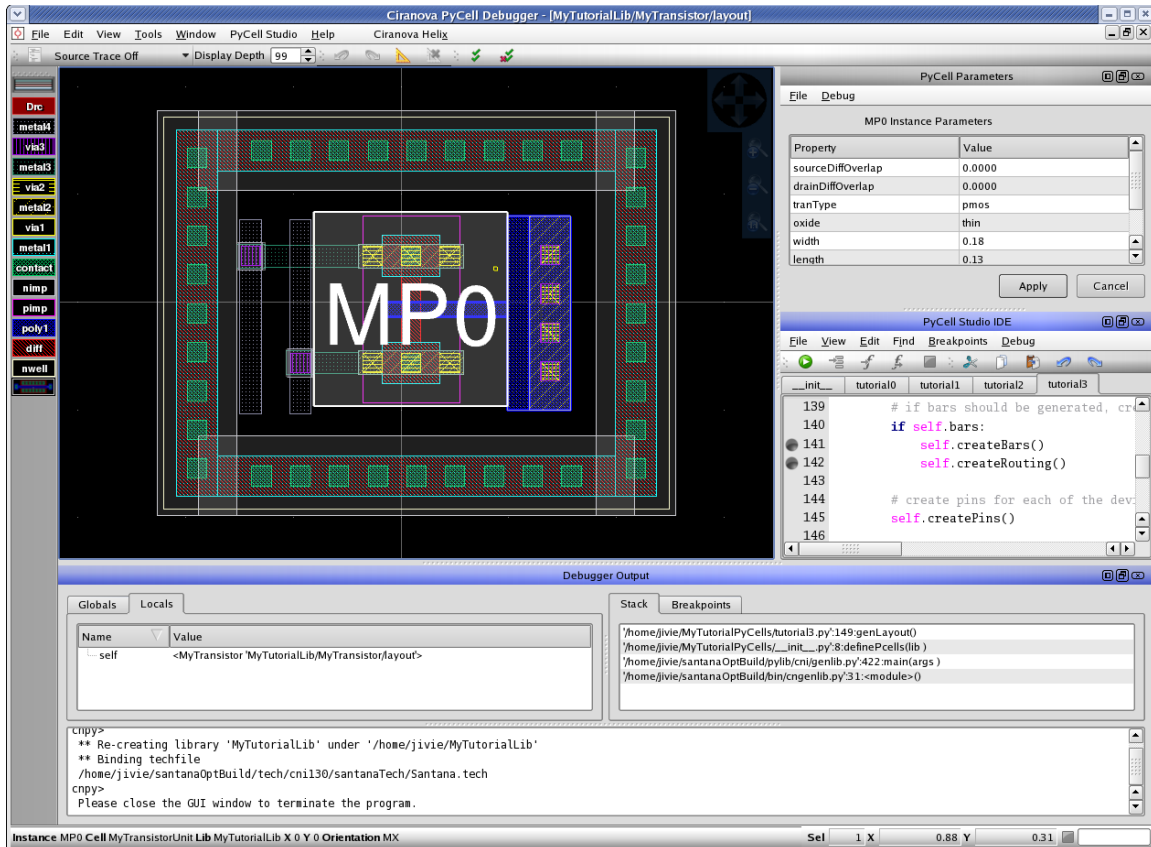
Step 6) Tracing Layout to Python Source Code

The MyTransistor PyCell just created contains several Bar layout objects, which were generated by the Python code for this PyCell. It is often useful to be able to examine the generated geometries in the layout and then trace back to the Python source code which generated these geometries. This is easily done using the Pyros “source tracing” feature. To see how this works, first make sure that the “PyCell Parameters” docking window is active; this can be done by clicking the right mouse button in the toolbar/menu area. Then turn on source tracing using the “Source Trace ON” pull-down icon on the main toolbar, and then click on the “Apply” button in the “PyCell Parameters” window. Select the leftmost Bar in the layout displayed in the main viewing window, and then use the *View>View Source Code for Selected Object* menu entry for the main **cndbg** window. This will then show the Python source code which generated this Bar in the “PyCell Studio IDE” code viewing window. This Python source code should then be displayed as follows:



Step 7) Change PyCell Parameters and observe the changes

The MyTransistor PyCell just created contains a PyCell instance called MyTransistorUnit that contains the basic geometries for the device. We can select the MyTransistorUnit within the MyTransistor PyCell and modify the parameters and update the instance to see the changes. From within the **cndbg** window left click over the gate area of the transistor. The MyTransistorUnit instance named “MP0” should now be selected. The PyCell parameters for this MyTransistorUnit instance should be displayed in the “PyCell Parameters” window, and these parameters should be displayed as follows:



Position the cursor in a textbox, select and change the text. For example, select the pmos in the tranType box and change it to nmos.

Click on the *Apply* button in the “PyCell Parameters” docking window to see the change in the layout. Using the *File>Restore Default Parameters* menu entry, and then clicking on the *Apply* button will restore the PyCell to its default parameters.

Click on the *File>Exit* menu entry from the **cndbg** window to exit the debugging tool.

Congratulations! You have now reached the end of the PyCell Studio Tutorial. We hope that you have learned about the PyCell Studio environment and how it can be used to create and debug Python Open Access PyCells.